Consistance Duale et réseaux non-binaires

Julien Vion

Consistance duale et réseaux non-binaires

Julien Vion

École des Mines de Nantes, LINA UMR CNRS 6241,
4, rue Alfred Kastler, FR-44307 Nantes, France.
julien.vion@emn.fr

Résumé

La Consistance Duale (DC) a été introduite par Lecoutre, Cardon et Vion dans [10, 11]. Il s'agit d'une nouvelle manière de gérer la Consistance de Chemin (PC), à la définition plus simple et permettant d'introduire de nouvelles approximations et algorithmes particulièrement efficaces en pratique. Un des points intéressants de cette définition est la possibilité de généraliser PC aux réseaux de contraintes (CNs) non-binaires, tout en conservant la nature des contraintes initiales du problème, et en exploitant leurs propagateurs. Cette généralisation permet notamment de voir la Consistance Duale/de Chemin comme un moyen simple et efficace de générer des contraintes binaires implicites directement à partir des propagateurs des contraintes du problème. Cet article montre les implications en termes de complexité de cette généralisation. Des résultats expérimentaux préliminaires montrent le potentiel pratique d'une génération à la volée de telles contraintes implicites, et met en évidence les faiblesses potentielles de la méthode. Des idées prospectives amenées à gérer ces faiblesses sont alors proposées.

Les consistances sont des propriétés des réseaux de contraintes (*Constraint Networks*, CNs), utilisées pour identifier et éliminer, généralement en temps et espace polynomiaux, des valeurs ou des instanciations globalement inconsistantes, c'est-à-dire ne pouvant appartenir aux solutions du CSP associé au CN. Elles sont essentielles dans le processus de résolution d'un CSP et une des principales raisons du succès de la programmation par contraintes [8, 2].

La consistance la plus utile est la consistance d'arc généralisée (GAC), qui permet de détecter des valeurs inconsistantes pour une contrainte donnée. La consistance de chemin (PC) est une des consistances les plus anciennes. Elle est définie sur les CNs binaires, et généralement appliquée aux graphes de contraintes complets [14]. La consistance duale (DC) est une définition alternative (équivalente), plus simple de PC. Elle a été utilisée dans [10] pour définir une nouvelle approximation de PC, nommée DC *conservative* (CDC), et dans [11] pour introduire de nouveaux algorithmes efficaces pour établir PC sur des graphes complets de contraintes binaires.

Dans cet article, nous proposons d'étendre la définition de (C)DC aux CNs non-binaires. DC peut être alors vue comme une technique simple pour déduire automatiquement des contraintes binaires implicites à partir des domaines et contraintes initiaux du CN. Les contraintes implicites sont des contraintes qui peuvent être ajoutées au CN sans en changer l'ensemble des solutions, mais permettant d'améliorer la propagation des décisions. Après les définitions et notations, nous rappelons les caractéristiques des algorithmes de GAC à gros grain, et présentons la consistance duale. Nous décrivons comment appliquer DC à des réseaux non-binaires de contraintes hétérogènes utilisant des propagateurs basés sur la sémantique des contraintes. Nous proposons l'algorithme sDC^{clone}, permettant d'établir DC sur des réseaux de contraintes quelconques en générant des contraintes implicites « à la volée », et qui exploite totalement l'incrémentalité des algorithmes de GAC sous-jacents pour obtenir une complexité temporelle dans le pire des cas en $O(ne_id^3 + nd\psi)$ ($O(n(e+e_i)d^3)$ pour des CNs binaires). Cet algorithme a une meilleure complexité que les algorithmes proposés précédemment[1]. sDC-2.1, une variante généralisée et optimisée de sDC-2 est ensuite proposée, et sa complexité étudiée.

Finalement, nous identifions expérimentalement les défauts de DC, pouvant empêcher son utilisation sur des problèmes industriels. En perspective de ces travaux, nous proposons des approximations de DC permettant de gérer ses faiblesses identifiées.

[1] ψ est la complexité amortie pour établir GAC sur le CN de manière incrémentale, et e_i est le nombre de contraintes implicites générées.

1 Préliminaires

1.1 CN et CSP

Un *réseau de contraintes* (CN) P consiste en un ensemble de n variables $\mathscr{X}$ et un ensemble de e contraintes $\mathscr{C}$. Un domaine, associé à chaque variable X et noté $\text{dom}^P(X)$, est un ensemble fini d'au plus d valeurs que la variable peut prendre dans le CN P. Quand ce sera possible sans ambigüité, nous noterons simplement $\text{dom}(X)$ au lieu de $\text{dom}^P(X)$. Les contraintes spécifient les combinaisons autorisées de valeurs pour des ensembles de variables donnés. Une instanciation I est un ensemble de couples variable/valeur, (X, v), notés X_v, avec v une valeur d'un univers donné U ($\forall X \in \mathscr{X}, \text{dom}(X) \subseteq U$). I est *valide* par rapport à un CN P ssi pour toute variable X impliquée dans I, $v \in \text{dom}^P(X)$.

Une *relation* R est un ensemble d'instanciations. Une *contrainte* C d'arité r est un couple $(\text{scp}(C), \text{rel}(C))$, avec $\text{scp}(C)$ un ensemble de r variables et $\text{rel}(C)$ une relation d'arité r. k est l'arité maximale des contraintes dans un CN donné. Pour une contrainte donnée C, une instanciation I de $\text{scp}(C)$ (ou d'un sur-ensemble de $\text{scp}(C)$, ne considérant dans ce cas que les variables de $\text{scp}(C)$) *satisfait* C ssi $I \in \text{rel}(C)$. On dit que I est *autorisée* par C. Pour contrôler si une instanciation satisfait une contrainte, on effectue un *test de contrainte*. Une instanciation I est *localement consistante* ssi elle est valide et autorisée par toutes les contraintes du CN. Une *solution* d'un CN $P(\mathscr{X}, \mathscr{C})$ est une instanciation localement consistante de toutes les variables de $\mathscr{X}$.

Un CSP est le problème de décision consistant à déterminer si une solution à un CN donné existe. Une instanciation est *globalement consistante* ssi elle est un sous-ensemble d'au moins une solution. Les instanciations qui ne sont pas globalement consistantes sont également appelées *no-goods*. Déterminer si une instanciation localement consistante est un no-good est NP-complet dans le cas général, mais les propriétés de consistance sont utilisées pour identifier des no-goods en utilisant des algorithmes polynomiaux. Étant donnée une consistance Φ, elle peut être *établie* sur P en utilisant un « algorithme de Φ-consistance », dont l'objectif est de détecter et supprimer toutes les instanciations Φ-inconsistantes jusqu'à l'obtention d'un point fixe (une *fermeture* de P par Φ est obtenue). Le plus souvent, le point fixe est unique. Le CN obtenu à partir de P en établissant la consistance Φ sera noté $\Phi(P)$.

1.2 Consistance d'arc généralisée

La consistance d'arc généralisée (GAC) est la propriété de consistance la plus commune et la plus utile.

Algorithm 1: GAC($P = (\mathscr{X}, \mathscr{C})$, $\mathscr{A}$)

P: le CN à filtrer
$\mathscr{A}$: un ensemble initial d'arcs à réviser

```
1 Q ← A
2 tant que Q ≠ ∅ faire
3     prendre (C, X) de Q
4     si revise(C, X) alors
5         si dom(X) = ∅ alors retourner ⊥
6         Q ← Q ∪ mod({X}, C)\(C, X)
7 retourner P
```

Il s'agit d'une propriété de *consistance de domaine* [2], c'est-à-dire qu'elle identifie des no-goods de taille 1 (des valeurs globalement inconsistantes). Les propriétés de consistance qui identifient des no-goods de plus grande taille sont appelées consistances de *relation*, puisqu'elles permettent de supprimer des instanciations des relations des contraintes.

Definition 1 (Consistance d'arc généralisée)**.** Étant donné un CN $P = (\mathscr{X}, \mathscr{C})$:

1. $X_a \mid X \in \mathscr{X}$ et $a \in \text{dom}(X)$ est GAC par rapport à la contrainte $C \in \mathscr{C}$ ssi $\exists I \mid X_a \in I \wedge I$ est valide et autorisée par C. I est alors appelée un *support* de a pour C.
2. X_a est GAC ssi $\forall C \in \mathscr{C} \mid X \in \text{scp}(C)$, X_a est GAC par rapport à C.
3. P est GAC ssi $\forall X \in \mathscr{X}$, $\forall a \in \text{dom}(X)$, X_a est GAC.

Notation 1. L'ensemble $\{(C, Y) \mid \{X, Y\} \subseteq \text{scp}(C) \wedge X \in \mathscr{X} \wedge C \in \mathscr{C}\}$ des arcs à réviser après une modification effectuée sur les variables de l'ensemble $\mathscr{X}$, ou sur les contraintes de l'ensemble $\mathscr{C}$, sera noté $\text{mod}(\mathscr{X}, \mathscr{C})$.

L'algorithme 1 présente la boucle principale des algorithmes de GAC à gros grain, i.e. des variantes de GAC-3. Les variantes résident dans la nature de la fonction `revise`, appelée à la ligne 4 de l'algorithme. Cette version de l'algorithme est « orientée arcs » : la file de propagation contient tous les arcs qui doivent être révisés. Un arc est un couple (C, X), avec $X \in \text{scp}(C)$. Dans un CN donné, on peut définir jusqu'à $O(ek)$ arcs. Un arc (C, X) doit être révisé s'il existe une possibilité pour X de ne pas être AC par rapport à C. Si l'on n'a aucune information sur l'état du CN, tous les arcs doivent être placés dans la file et être ainsi révisés au moins une fois. Si l'on sait qu'une unique variable X a été modifiée dans un CN GAC, seuls les arcs $\text{mod}(\{X\}, \mathscr{C})$ doivent être insérés. L'algorithme 1 peut être initialisé à partir d'un ensemble d'arcs $\mathscr{A}$ (ligne 1). En utilisant la notation 1, GAC

peut être établie sur un CN donné $P = (\mathscr{X}, \mathscr{C})$ par un appel $P \leftarrow \text{GAC}(P, \text{mod}(\mathscr{X}, \mathscr{C}))$.

La complexité dans le pire des cas de ces algorithmes vient du fait qu'un arc donné (C, X) n'est inséré dans la file de propagation que lorsque l'une des $O(k)$ variables de scp(C) est modifiée (il faut alors contrôler si les supports de X par rapport à C sont toujours valides). Or, une variable ne peut être modifiée que d fois. Ainsi, $O(ek^2d)$ appels à `revise` peuvent être effectués dans le pire des cas. L'algorithme de base pour `revise` effectue une itération sur le produit Cartésien des domaines des variables de scp(C) jusqu'à trouver un support pour chaque valeur des domaines, d'où une complexité en $O(kd^k)$ pour `revise` (on suppose qu'un test de contrainte se fait en $O(k)$), et de $O(ek^3d^{k+1})$ pour établir GAC. L'idée de GAC-2001 [5] est de rendre la fonction `revise` incrémentale, de sorte que la complexité amortie de tous les appels à `revise` pour une contrainte donnée est en $O(k^2d^k)$,[2] d'où une complexité en $O(ek^2d^k)$. GAC-schema [4] atteint la complexité optimale $O(ekd^k)$ en utilisant des files de propagation à *grain fin* et en exploitant la multi-directionnalité des contraintes. Cependant, cela nécessite des structures de données additionnelles, et l'algorithme n'est pas toujours plus performant en pratique. D'autres travaux ont indiqué que d'autres variantes sous-optimales de GAC-3, comme GAC-3^{rm} [12] ou GAC-watched [9] sont les plus efficaces en pratique quand ils sont maintenus au cours de la recherche, grâce à l'exploitation de structures de données *stables au backtrack*. Dans le cas binaire ($k = 2$), (G)AC-2001 est optimal en $O(ed^2)$. Finalement, dans le cas binaire, il existe des algorithmes hautement optimisés comme AC-3^{bit} [13], permettant la propagation efficace d'un grand nombre de contraintes binaires en extension.

Toutes les variantes des algorithmes de GAC sont incrémentales : une fois le point fixe atteint dans un CN donné, la complexité amortie dans le pire des cas pour plusieurs appels à l'algorithme, en supprimant au moins une valeur entre chaque appel, est la même que la complexité d'un seul appel. Pour appliquer GAC-3 incrémentalement, il faut veiller à ne réviser un arc que si et seulement si une valeur a été supprimée dans une variable impliquée par la contrainte de l'arc.

1.3 Propagateurs

L'idée des *propagateurs* provient du schéma de propagation générique AC-5 [19]. Les algorithmes de GAC génériques présentés dans la section précédente sont tous exponentiels en l'arité des contraintes, et ne sont pas applicables pour des contraintes de grande arité. Cependant, en pratique, on peut souvent établir GAC efficacement pour une contrainte non-binaire en exploitant les propriétés sémantiques de celle-ci [1, 16]. AC-5 permet ceci en *abstrayant* la fonction `revise`, qui peut alors être spécialisée pour chaque type de contrainte. Ces fonctions `revise` spécialisées sont souvent appelées *propagateurs*. Dans ce contexte, les algorithmes de GAC généraux sont le plus souvent utilisés pour propager des contraintes définies en *extension*, c'est-à-dire à partir d'une liste exhaustive d'instanciations interdites[3], ou pour des contraintes définies en *intention* et utilisant des opérateurs scalaires.

La plupart des travaux sur la résolution générique des CSP se sont focalisés sur l'utilisation de contraintes homogènes, le plus souvent binaires en extension. En pratique, les problèmes industriels mettent en jeu des contraintes hétérogènes, associées à des propagateurs efficaces basés sur leur sémantique. Dans cet article, nous ne faisons aucune hypothèse sur la manière dont GAC doit être établie. Toutes les contraintes pour lesquelles il existe un propagateur spécifique sont conservées, et la sémantique de celles-ci est exploitée.

Par la suite, nous noterons $O(\phi)$ la complexité dans le pire des cas pour établir GAC sur un réseau de contrainte donné, $O(\psi)$ la complexité amortie pour établir GAC *incrémentalement* sur ce réseau, en supprimant au moins une valeur entre chaque propagation, et $O(\rho)$ la complexité spatiale totale du réseau et de tous les propagateurs impliqués. Dans un CN général utilisant GAC-schema comme algorithme de propagation, $O(\phi) = O(\psi) = O(ekd^k)$ et $O(\rho) = O(nd + ekd)$.

1.4 Consistance duale

La consistance duale (DC) introduite dans [10], est une définition alternative (équivalente) à celle de la consistance de chemin [14] :

Definition 2 (Consistance duale)**.** Étant donné un CN $P = (\mathscr{X}, \mathscr{C})$, une instanciation binaire $I = \{X_a, Y_b\}$ est DC ssi $X_a \in \text{AC}(P|_{Y=b}) \wedge Y_b \in \text{AC}(P|_{X=a})$.

P est DC ssi toutes les instanciations binaires localement consistantes de P sont DC.

P est fortement DC (sDC) ssi il est à la fois AC et DC.

L'algorithme connu le plus efficace pour établir la consistance duale forte est sDC-2, décrit dans [11]. Lecoutre *et al.* montrent qu'établir sDC avec cet algorithme a une complexité temporelle dans le pire des cas en $O(n^5d^5)$, nettement supérieure aux meilleures

[2] Le facteur additionnel k provient du fait que la multi-directionnalité des relations ne peut être exploitée, et chaque instanciation peut être prise en compte jusqu'à k fois.

[3] Dans le cas d'une liste d'instanciations *autorisées*, on peut utiliser d'autres algorithmes comme STR [17].

complexités connues pour PC, soit $O(n^3d^3)$. Cependant, le pire des cas pour sDC-2 n'a que très peu de chances d'apparaitre, et l'algorithme est en pratique plus rapide que les algorithmes de PC état-de-l'art sur la plupart des instances de CSP. De plus, sDC-2 n'utilise qu'une structure de données très légère (en $O(n)$).

2 Consistance duale et réseaux non-binaires : algorithmes et complexités

Les algorithmes existants pour PC nécessitent que le CN soit un graphe complet de contraintes binaires, supportant la composition. La définition de DC permet d'établir les remarques suivantes :

1. En remplaçant simplement AC par GAC dans la définition 2, DC peut être appliquée à un CN d'arité quelconque.
2. Les algorithmes de DC ne font aucune hypothèse sur la manière dont (G)AC doit être établi. Les algorithmes peuvent donc utiliser tous les propagateurs état-de-l'art (basés sur la sémantique ou pour les contraintes en extension) fournis par le solveur.
3. Pour établir sDC, il « suffit » d'être en mesure de stocker et exploiter des no-goods binaires, et ce même si le CN original inclut des contraintes non-binaires. Ceci ne suppose pas nécessairement de générer un graphe complet de contraintes (seulement dans le pire des cas).[4]

La dernière proposition provient d'un nouveau point de vue sur DC (et PC) : les contraintes additionnelles introduites pour établir DC sont des contraintes qui sont *impliquées* par les contraintes initiales du problème. DC nous permet de déduire ces contraintes de manière purement sémantique, en exécutant les propagateurs des contraintes d'origine.

Trois algorithmes différents, sDC-1, sDC-2 et sDC-3, établissant la consistance duale forte sur des graphes complets de contraintes binaires, sont décrits dans [11]. Nous proposons d'étendre sDC-1 et sDC-2 pour prendre en compte les réseaux de contraintes non-binaires (sDC-3 est spécifiquement destiné aux réseaux binaires et les résultats expérimentaux montrent qu'il s'agit de l'algorithme le moins intéressant en pratique). Nous exploitons l'incrémentalité des algorithmes de propagation pour obtenir une meilleure borne théorique de complexité temporelle. Nous nous inspirons pour cela de l'algorithme SAC-OPT [3]. Les algorithmes de sDC (et SAC) fonctionnent à partir des *tests de singleton* : pour chaque valeur X_v d'un CN

[4] C'est également vrai pour les algorithmes de PC, mais cela n'a à notre connaissance pas été expérimenté dans des travaux antérieurs.

Algorithm 2: sDC-1($P = (\mathscr{X}, \mathscr{C})$)

```
1  P ← GAC(P, mod(𝒳, 𝒞))
2  marque ← X ← first(𝒳)
3  tant que X = marque faire
4      si |dom(X)| > 1 ∧ checkVar-1(P, X) alors
5          P ← GAC(P, mod({X}, 𝒞))
6          si P = ⊥ alors retourner ⊥
7          marque ← X
8      X ← next(𝒳, X)
9  retourner P
```

Algorithm 3: checkVar-1($P = (\mathscr{X}, \mathscr{C})$, X)

```
1  modif ← faux
2  pour ch. a ∈ dom(X) faire
3      P' ← GAC(P|_{X=a}, mod({X}, 𝒞))
4      si P' = ⊥ alors
5          retirer a de dom^P(X)
6          modif ← vrai
7      sinon
8          pour ch. Y ∈ 𝒳\X faire
9              soit C t.q. C ∈ 𝒞_i ∧ scp(C) = {X, Y}
10             pour ch. b ∈ dom^P(Y) | b ∉
               dom^{P'}(Y) ∧ {X_a, Y_b} ∈ rel(C) faire
11                 retirer {X_a, Y_b} de rel(C)
12                 modif ← vrai
13 retourner modif
```

P, v est affectée à X ($\mathrm{dom}(X)$ est réduit au *singleton* $\{v\}$), et (G)AC est établi sur le CN résultant, noté $P|_{X=v}$. SAC supprime la valeur du domaine de la variable ssi une inconsistance est détectée. DC va plus loin en enregistrant des informations pouvant être déduites du test de singleton sous forme de no-goods binaires qui sont ajoutés au CN en modifiant et/ou en insérant des contraintes implicites. Le sous-ensemble des contraintes implicites sera noté $\mathscr{C}_i$ ($\mathscr{C}_i \subseteq \mathscr{C}$). Toutes les contraintes binaires en extension (ou pouvant efficacement être converties en contraintes en extension) du réseau initial peuvent immédiatement être placées dans $\mathscr{C}_i$. On notera $e_i = |\mathscr{C}_i|$. On a $e_i \leq \binom{n}{2} \in O(n^2)$. D'après la preuve dans [3] sur la globalité des supports pour SAC, toute modification effectuée lors d'un test de singleton doit entrainer le contrôle de tous les singletons impliquant les autres variables.

2.1 sDC-1 : l'approche naïve.

sDC-1, déjà décrit dans [11], est l'algorithme le plus « naïf » pour établir sDC. Les algorithmes 2 et 3 dé-

crivent sDC-1. L'algorithme itère sur toutes les valeurs des domaines des variables. La *marque* et les fonctions `first`/`next` utilisées dans l'algorithme 2 sont utilisées pour parcourir les variables du CN en boucle, jusqu'à ce que toutes les variables aient été testées par `checkVar-1` sans qu'aucune modification n'ait été effectuée. `next`$(\mathscr{X}, X)$ considère $\mathscr{X}$ comme un ensemble ordonné, et renvoie soit la variable juste après X dans $\mathscr{X}$, soit la première variable de $\mathscr{X}$ (`first`$(\mathscr{X})$), ssi X était la dernière variable de $\mathscr{X}$. Les tests de singleton sont effectués dans la boucle principale **pour ch. faire** de `checkVar-1` (algorithme 3). La seconde boucle, lignes 8-12 permet de stocker les no-goods pouvant être déduits à partir de l'établissement de GAC à la ligne 3. À la ligne 9, la contrainte est créée « à la volée » ssi elle n'existe pas. Les contraintes créées sont initialement des contraintes universelles, qui autorisent toutes les instanciations des variables.

Proposition 1. Appliqué à un CN quelconque, sDC-1 a une complexité temporelle dans le pire des cas en $O(ne_i^2d^5 + ne_id^3\phi)$ et une complexité spatiale en $O(e_id^2)$.

Ébauche de preuve. $O(nd)$ singletons doivent être testés, et si n'importe quel test de singleton entraine une modification, tous les singletons doivent à nouveau être testés [3]. La plus petite modification pouvant survenir est la suppression d'un no-good binaire dans la relation d'une contrainte implicite. Il peut ainsi y avoir jusqu'à $O(e_id^2)$ modifications. Un test de singleton consiste en :

1. L'établissement de GAC en $O(e_id^2 + \phi)$ sur le CN (ligne 3 de l'algorithme 3). Le terme $O(e_id^2)$ correspond à la propagation des e_i contraintes implicites binaires supplémentaires, à l'aide d'un algorithme d'AC optimal.
2. Traiter les $O(nd)$ modifications (valeurs supprimées), et supprimer les no-goods correspondants dans les contraintes implicites (lignes 8-12 de l'algorithme 3). L'exploitation des deltas des domaines modifiés par une propagation (comme suggéré par le schéma de propagation AC-5) permet de rendre ce travail incrémental, et la complexité amortie obtenue est en $O(e_id^2)$, qui peut être ignorée.

La seule structure de données utilisée par sDC-1 correspond au stockage des no-goods dans les contraintes binaires en extension, en $O(e_id^2)$. Les structures de données de l'algorithme d'AC optimal utilisé pour propager ces contraintes sont en $O(e_id)$ et peuvent être négligées. □

Les résultats expérimentaux décrits dans [11] indiquent que malgré la simplicité et la complexité théorique dans le pire des cas élevée, cet algorithme se comporte très bien en pratique, le plus souvent mieux que les algorithmes de PC état-de-l'art.

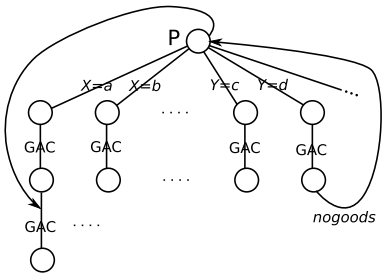


FIG. 1 – Illustration de l'incrémentalité de GAC sur les tests de singleton

2.2 sDC^{clone} : établir sDC en $O(ne_id^3 + nd\psi)$.

Cette complexité pour établir DC est obtenue en exploitant complètement l'incrémentalité des algorithmes de (G)AC. Comme illustré par la figure 1, si des no-goods sont déduits du test de singleton $Y = d$, le test de singleton antérieur $X = a$ ne devrait pas être recalculé entièrement, mais relancé incrémentalement à partir du dernier point fixe obtenu pour $X = a$. L'idée de sDC^{clone}, empruntée à l'algorithme SAC-OPT, est d'obtenir ce résultat en créant une copie physique du CN en mémoire pour chacun des $O(nd)$ tests de singleton possibles. Afin de bénéficier de l'incrémentalité des algorithmes de propagation, toutes les structures de données de ces algorithmes doivent également être dupliquées.

Il faut remarquer que les contraintes implicites doivent être propagées *à chaque fois qu'elles sont modifiées*, et non pas seulement quand une valeur est supprimée du domaine d'une variable. En consultant l'algorithme 1, on constate que chaque arc impliquant une contrainte implicite peut être inséré $O(d^2)$ fois, ce qui signifie $O(e_id^2)$ appels à `revise`. Bien que la procédure `revise` d'AC-2001 soit incrémentale, elle requiert au moins $O(d)$ opérations pour valider les supports courants. La complexité d'AC-2001 devient donc $O(e_id^3)$, et cet algorithme ne peut être utilisé pour obtenir une complexité optimale pour propager les contraintes implicites : il faut utiliser un algorithme à *grain fin* comme AC-4, AC-6 ou AC-7.

Proposition 2. Appliqué à un CN quelconque, sDC^{clone} admet une complexité temporelle dans le pire des cas en $O(e_ind^3 + nd\psi)$ et une complexité spatiale en $O(e_ind^2 + nd\rho)$.

Ébauche de preuve. La complexité temporelle dans le pire des cas est obtenue en considérant entièrement l'incrémentalité de GAC pour chacun des $O(nd)$ tests

de singleton, en supposant une complexité amortie pour établir GAC incrémentalement sur le CN en $O(\psi)$, et $O(e_i d^2)$ pour établir AC sur les contraintes implicites, en utilisant un algorithme d'AC optimal à grain fin.

Le CN doit être cloné $O(nd)$ fois. Cependant, la liste des no-goods (i.e. les relations des contraintes implicites) peut être partagé entre tous les clones. Seules les structures de données de l'algorithme d'AC optimal (en $O(e_i d)$) doivent être clonées. On en déduit la complexité spatiale de l'algorithme. □

Par exemple, pour un CN consistant uniquement en une série de contraintes *all-diff* ($O(\psi) = O(ek^2d^2)$ et $O(\rho) = O(ekd)$ [16]), par exemple pour modéliser un problème de carré latin, sDC peut être établie avec cet algorithme en $O(e_i nd^3 + enk^2d^3)$ avec une complexité spatiale en $O(e_i nd^2 + enkd^2)$.

Dans le cas binaire, sDC^{clone} atteint une complexité spatiale dans le pire des cas en $O(n(e + e_i)d^3)$, c'est-à-dire une complexité inférieure aux algorithmes de PC connus[5]. Cependant, les besoins en termes d'espace restent excessifs pour une utilisation pratique. De plus, Debruyne & Bessière indiquent dans [3] que les résultats expérimentaux conduits autour de l'algorithme SAC-OPT, construit suivant le même principe, se sont révélés décevants. L'algorithme n'est donc pas décrit ni expérimenté plus avant dans cet article.

2.3 sDC-2.1 : un compromis

« Restaurer l'état du CN, » une opération essentielle pour obtenir les complexités améliorées de sDC^{clone} ou SAC-OPT, peut être réalisé de manière très naturelle, sans aucune structure de données additionnelle, lors de l'établissement de DC. L'information enregistrée dans les contraintes implicites lors du précédent test du même singleton peut être exploitée pour restaurer les domaines des variables dans l'état dans lesquels ils étaient à la fin du précédent test. Ceci peut être effectué par un appel à la fonction `revise` des contraintes implicites impliquant la variable du singleton courant X_a : **pour ch.** $(C, Y) \in \text{mod}(\{X\}, \mathscr{C}_i)$ **faire** `revise`(C, Y). Ce processus est un *Forward Checking* et a une complexité dans le pire des cas en $O(nd)$, puisqu'il peut y avoir au plus n contraintes binaires impliquant une variable donnée, et que l'appel à `revise` sur une contrainte binaire dont l'une des variables est un singleton est en $O(d)$.

Cette observation avait été en partie exploitée pour concevoir l'algorithme sDC-2. Nous présentons ici sDC-2.1, une version optimisée et étendue de sDC-2, gérant les contraintes implicites et non-binaires de manière spécifique. L'algorithme est décrit par les algorithmes 4 et 5. La structure $lastModified$ de sDC-2 est remplacée par une nouvelle structure $arcQueue$ et par $O(nd)$ pointeurs (un par singleton X_a), notés $firstArc[X_a]$. Cette structure de données fonctionne comme suit : $arcQueue$ contient tous les arcs qui doivent être révisés lors de tous les tests de singleton. Un arc est inséré à la fin de la file lorsqu'une variable voisine ou la contrainte elle-même est modifiée. À chaque fois qu'un arc de $arcQueue$ est révisé pour le test de singleton X_a, $firstArc[X_a]$ est déplacé vers l'élément suivant de la file. Lors de l'appel à GAC, la file de propagation est initialisée avec tous les arcs de $arcQueue$ en commençant par $firstArc[X_a]$. Ce mécanisme assure que (1.) l'ajout d'un arc à la file de propagation se fait en temps constant tout en pouvant être géré par tous les singletons, et (2.) que les initialisations des files de propagation se font de manière incrémentale.

Ces structures de données sont utilisées à la ligne 5 de l'algorithme 4 et à la ligne 9 de l'algorithme 5 pour initialiser les files de propagation de l'algorithme sous-jacent, de sorte d'assurer que seuls les propagateurs susceptibles de réaliser des filtrages seront appelés (i.e. une variable impliquée par la contrainte ou la contrainte elle-même a été modifiée depuis le dernier test de ce singleton). X', Y', C' et $\mathscr{C}'_i$ sont les variables, contraintes ou ensembles de contraintes de P' qui correspondent respectivement à X, Y, C et $\mathscr{C}_i$ dans P.

Algorithm 4: sDC-2.1$(P = (\mathscr{X}, \mathscr{C}), \mathscr{C}_i)$

```
1  P ← GAC(P, mod(𝒳, 𝒞))
2  mark ← X ← first(𝒳)
3  répéter
4      si |dom(X)| > 1 ∧ checkVar-2.1(P, 𝒞_i, X)
       alors
5          P' ← GAC(P, {arcQueue[i] | firstArcMain ≤ i < cnt})
6          si P' = ⊥ alors retourner ⊥
7          pour ch. (Y, b) | Y ∈ 𝒳 ∧ b ∈ dom^P(Y) ∧ b ∉ dom^P'(Y) faire
8              retirer b de dom^P(Y)
9              pour ch. (C, Z) ∈ mod({Y}, 𝒞) faire
10                 arcQueue[cnt++] ← (C, Z)
11         firstArcMain ← cnt
12         mark ← X
13     X ← next(𝒳, X)
14 jusqu'à X = mark
15 retourner P
```

[5] Bien sûr, le nombre e_i de contraintes implicites ne peut être connu avant l'appel à l'algorithme, et peut atteindre $\binom{n}{2}$.

Algorithm 5: checkVar-2.1($P = (\mathscr{X}, \mathscr{C})$, $\mathscr{C}_i$, X)

```
1  modif ← faux
2  pour ch. a ∈ dom(X) faire
3      si cnt ≤ |𝒳| alors
           /* Premier tour                                   */
4          P' ← GAC(P|_{X=a}, mod({X}, 𝒞))
5      sinon
           /* Tours suivants : forward
              checking                                       */
6          P' ← P|_{X=a}
7          pour ch. (C', Y') ∈ mod({X'}, 𝒞_i') faire
8              revise(C', Y')
           /* Propagation avec les files de
              propagation préinitialisées                    */
9          P' ←
           GAC(P', {arcQueue[i] | firstArc[X_a] ≤
           i < cnt}))
10     si P' = ⊥ alors
11         retirer a de dom^P(X)
12         pour ch. (C, X) ∈ mod({X}, 𝒞) faire
13             arcQueue[cnt++] ← (C, X)
14         modif ← vrai
15     sinon
16         pour ch. Y | {X, Y} ⊆ 𝒳 faire
17             soit C t.q. C ∈ 𝒞_i ∧ scp(C) = {X, Y}
18             pour ch. b ∈ dom^P(Y) | b ∉
               dom^{P'}(Y) ∧ {X_a, Y_b} ∈ rel(C) faire
19                 retirer {X_a, Y_b} de rel(C)
20                 arcQueue[cnt++] ← (C, X)
21                 arcQueue[cnt++] ← (C, Y)
22                 modif ← vrai
23 firstArc[X_a] ← cnt
24 retourner modif
```

Proposition 3. Appliqué à un réseau de contraintes quelconque, sDC-2.1 a une complexité temporelle dans le pire des cas en $O(e_i^2 d^4 + e_i n d^5 + nkd^2\phi)$ et une complexité spatiale en $O(e_i d^2 + ek^2 d)$.

Ébauche de preuve. Dans cette variante de l'algorithme, un test de singleton consiste en :

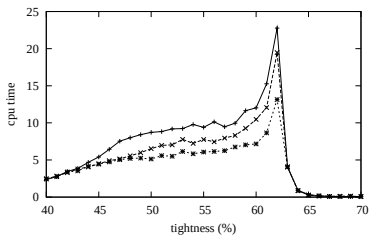
1. La restauration de l'état du CN à la fin du test de singleton précédent du même couple variable/valeur, en $O(nd)$ par Forward Checking. Le Forward Checking sur tous les singletons est amorti en $O(e_i d^2)$, et est réalisé $O(e_i d^2)$ fois dans le pire des cas, s'agissant du nombre de modifications possibles. D'où un premier terme en $O(e_i^2 d^4)$.
2. L'initialisation de la file de propagation de GAC. Dans le pire des cas, chacun des $O(e_i + ek)$ arcs est inséré une fois à chaque modification possible de chaque test de singleton ($O(d^2)$ pour les arcs impliquant les contraintes implicites et $O(kd)$ pour les autres), d'où une complexité amortie en $O(nd.(e_i d^2 + ek^2 d))$.
3. La propagation de l'assignation ou des mises à jour. Puisque les structures de données des propagateurs ne sont pas clonées, l'incrémentalité de ceux-ci est perdue, et retombe à $O(e_i d^2 + \phi)$. On sait qu'un propagateur donné ne peut être appelé que $O(kd)$ fois pour un singleton donné (resp. $O(d^2)$ fois pour les contraintes implicites), quand une valeur (resp. un no-good) est supprimée. Pour chacun des $O(nd)$ singletons, la complexité amortie pour toutes les propagations d'un test de singleton est donc en $O(e_i d^4 + kd\phi)$.
4. Si une inconsistance est détectée, la suppression d'une valeur et la mise à jour de *arcQueue* (ligne 13 de l'algorithme 5). La détection d'une inconsistance à la ligne 10 ne peut survenir que $O(nd)$ fois et nécessite $O(n)$ opérations pour la mise à jour d'*arcQueue*, qui sont amorties pour un total en $O(e_i d + ed)$. On peut négliger ce terme en considérant $e \leq \phi$.
5. Sinon, la mise à jour des contraintes implicites et de *arcQueue*. Comme pour sDC-1, ces opérations sont incrémentales si les deltas sont exploités, et peuvent être négligées.

En ce qui concerne la complexité spatiale : les relations des contraintes implicites sont en $O(e_i d^2)$. Les $O(e_i)$ arcs correspondant aux contraintes implicites peuvent être insérés $O(d^2)$ fois dans la file de révision, et les $O(ek)$ arcs correspondant aux contraintes initiales peuvent être insérées $O(kd)$ fois. La structure *arcQueue* a donc une complexité spatiale en $O(e_i d^2 + ek^2 d)$ et les pointeurs $firstArc$ en $O(nd)$. Ce dernier terme peut être négligé, en considérant $\rho \geq nd$. □

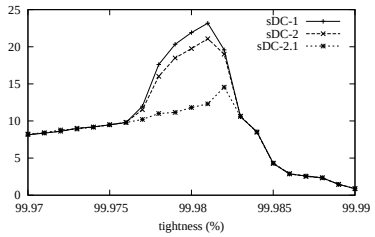
Appliqué à un graphe complet de contraintes binaires ($O(\phi) = O(n^2 d^2)$ et $O(e_i) = O(n^2)$), la complexité de l'algorithme devient $O(n^4 d^4 + n^3 d^5)$, ce qui présente une amélioration par rapport à sDC-2 (en $O(n^5 d^5)$). Si l'on considère par exemple un CN composé initialement de contraintes *all-diff*, pouvant être propagées en $O(k^{1.5} d^2)$ [16], on peut ainsi établir sDC en $O(e_i^2 d^4 + e_i n d^5 + enk^{2.5} d^4)$.

2.3.1 Notes d'implémentation

En pratique, la structure *arcQueue* peut sans risque être remplacée par deux tableaux d'entiers



$(50, 50, 1225, 2)$



$(20, 6, 50, 8)$

FIG. 2 – Prétraitement sur des instances aléatoires présentant des contraintes de dureté variable. Temps CPU en secondes. Les caractéristiques des problèmes sont données sous forme de quadruplets (n, d, e, k).

$lastModVar$ et $lastModCons$, c'est-à-dire une structure plus proche de l'algorithme sDC-2 original. L'initialisation des files de propagation avant chaque propagation n'est alors plus incrémentale et ajoute un terme théorique en $O(e_i n^2 d^3 + e_i^2 n d^3)$ à la complexité temporelle dans le pire des cas de l'algorithme. Cependant, la complexité spatiale est moindre ($O(n + e_i)$ au lieu de $O(nd + e_i d^2 + ekd)$), et le résultat est plus rapide en pratique, cette structure permettant de regrouper naturellement plusieurs révisions d'un même arc.

L'algorithme de GAC « orienté arcs » présenté à l'algorithme 1 peut être émulé par une version « orientée variables » améliorée, exploitant des heuristiques d'ordonnancement des révisions et des compteurs spéciaux pour éviter les révisions inutiles, comme décrit dans [6, 7].

3 Expérimentations

Les expérimentations sont effectuées sur des *benchmarks* de la Troisième Compétition Internationale de Solveurs [18], sur des problèmes binaires et non-binaires, impliquant des contraintes en intention, extension ou présentant des propagateurs spécifiques connus. Nous avons utilisé le solveur Concrete, construit à partir de la bibliothèque de programmation CSP4J [20].

La figure 2 donne une idée de l'amélioration apportée par sDC-2.1 par rapport à sDC-1 et sDC-2, même sur des problèmes binaires. sDC est établie sur des problèmes binaires (graphe du haut) et non-binaires (graphe du bas) aléatoires, avec chacun des algorithmes. Les différences entre les algorithmes apparaissent près du point de transition, quand les singletons doivent être parcourus plusieurs fois pour atteindre le point fixe.

Deux stratégies sont comparées : les problèmes sont résolus en utilisant un algorithme MGAC-*dom/wdeg* classique, après une phase de préparation où GAC ou sDC sont respectivement établis. Les structures de données et les algorithmes de propagation utilisés pour les contraintes binaires en extension sont ceux d'AC-3^{bit} ou AC-3^{bit+rm}.

La table 1 montre des résultats représentatifs d'instances pour lesquelles l'application de sDC a un impact positif sur la recherche. sDC est particulièrement efficace sur les problèmes très difficiles, où le nombre réduit de nœuds explorés parvient à compenser le temps passé durant le prétraitement et celui passé à propager les contraintes supplémentaires pendant la recherche (ce temps peut être estimé par la métrique *nœuds par seconde*). Les problèmes sélectionnés sont des problèmes structurées présentant différents types de contraintes. Les instances *taillard* sont des problèmes d'ordonnancement d'atelier, modélisés par des contraintes d'*inégalité disjonctive* avec un propagateur incrémental spécifique à complexité linéaire. *tsp* est un problème de voyageur de commerce et *series* un problème de séries d'intervalles impliquant des contraintes de tables positives ternaires, propagées avec un algorithme STR [17]. *scen11-f1* est l'instance de problème d'allocation de fréquences RFLAP la plus difficile. Bien que cette instance n'implique que des contraintes binaires, elle montre qu'il n'est pas nécessaire de compléter le graphe de contraintes pour établir DC (*scen11-f1* utilise 680 variables et compléter le graphe nécessiterait plus de 230 000 contraintes).

Dans de nombreux cas, les nouvelles contraintes interfèrent avec l'heuristique de choix de variable : les meilleures heuristiques de choix de variable génériques, comme *dom/wdeg*, exploitent en effet la structure du CN pour sélectionner les variables à affecter. L'instance *tsp-25-715* est influencée de manière spectaculaire par ce phénomène.

La table 2 montre des exemples d'instances pour lesquelles l'application de sDC est contre-productive. Trois principaux inconvénients peuvent être identifiés, et une instance représentative de chaque cas a été choise : pour *os-taillard-7-95-7*, bien que le temps de prétraitement soit très raisonnable, la réduction du nombre de nœuds ne permet pas de compenser le temps perdu à propager les nouvelles contraintes. Dans le cas de *os-gp-10-10*, un très grand nombre de contraintes implicites est généré (jusqu'à 5 000 pour

		GAC	sDC	GAC	sDC
		tsp-25-715 (76, 1 001, 350, 3)		*scen11-f1* (680, 42, 4 103, 2)	
prepro	cpu	0	353	0	41
	add cstr	0	2 303	0	757
search	cpu	497	**23**	9 646	**9 408**
	nodes	397 k	**2k**	6 749 k	**4 335 k**
	nodes/s	**798**	96	**700**	461
total	cpu	497	**376**	9 646	**9 448**
		series-15 (29, 15, 210, 3)		*os-taillard-7-100-0* (49, 434, 294, 2)	
prepro	cpu	0	1	0	27
	add cstr	0	182	0	294
search	cpu	475	**238**	> 600	**104**
	nodes	6 920 k	**1 890 k**	> 2 790 k	**148 k**
	nodes/s	**15 142**	7 908	**4 650**	1 423
total	cpu	475	**239**	> 600	**131**

Tab. 1 – Comparaison des performances de la recherche avec et sans prétraitement par sDC. Instances positives représentatives.

		GAC	sDC	GAC	sDC	GAC	sDC
		os-taillard-7-95-7 (49, 403, 294, 2)		*os-gp-10-10-1092* (101, 1 090, 1 000, 2)		*graceful-K5-P2* (35, 26, 370, 3)	
prepro	cpu	0	25	0	> 275	0	5
	add cstr	0	303	0	> 1 020	0	200
search	cpu	**675**	1,654	> 2 400	–	**157**	246
	assgn	2 749 k	**1,698 k**	> 864 k	–	**985 k**	993 k
	nodes/s	**4 073**	1,026	360	–	**6 280**	4 040
total	cpu	**675**	1,679	> 2 400	–	**157**	251

Tab. 2 – Instances représentatives négatives.

des tailles de domaines de l'ordre de 1 000 valeurs), et elles ne peuvent alors plus être stockées en mémoire (dans une limite de 600 Mio). Finalement, dans le cas de l'instance de *graceful*, *dom/wdeg* est perturbé par les nouvelles contraintes. Nous avons cependant constaté que ce cas reste assez rare, et d'autres stratégies de recherches plus proches de la sémantique des problèmes ne seraient bien sûr pas affectées.

		GAC	sDC/RC
		os-gp-10-10-1092	
prepro	cpu		799
	add cstr		851
search	cpu		599
	nodes		321k
	nodes/s		536
total	cpu		1,398

Tab. 3 – Résoudre un problème en générant des contraintes convexes par rangée implicites

4 Perspectives

Le principal intérêt de la consistance duale est de pouvoir automatiquement améliorer la robustesse des solveurs face à des modèles naïvement modélisés. En effet, certaines des contraintes implicites introduites par DC pourraient être insérées manuellement pendant la phase de modélisation, avec un algorithme de propagation approprié et des structures de données plus légères.

Cependant, comme les contraintes ajoutées par la consistance duale sont des contraintes implicites, elles peuvent être facilement relâchées sans ajouter de solutions au CSP. La DC conservative proposée dans [10] est déjà un bon exemple d'une telle approximation : les no-goods qui ne peuvent être ajoutés aux contraintes originellement présentes dans le problème sont ignorés. En perspective à nos travaux, nous proposons ces extensions :

– Relâcher les contraintes implicites pour qu'elles respectent une certaine propriété, comme la convexité par rangées (RC). De telles propriétés permettent de propager plus efficacement les contraintes implicites. La table 3 donne un

exemple de problème difficile résolu par l'ajout de contraintes convexes RC grâce à DC (les nogoods qui violent la propriété sont ignorés). La DC complète n'avait pas pu être établie sur le même problème (cf table 2).
– Ne propager les contraintes implicites que si elles sont actives pendant la recherche. On pourra se référer à des travaux tels que [15] pour cela, sans avoir à détecter les contraintes redondantes.

5 Conclusion

Dans cet article, nous avons proposé une extension de la définition de la consistance duale, et donc de la consistance de chemin, aux réseaux de contraintes non-binaires. Nous avons montré comment utiliser DC pour déduire automatiquement des contraintes binaires implicites à partir des domaines et contraintes originaux du CN. Une nouvelle variante d'algorithme de consistance duale forte, nommée sDC^{clone}, présentant une complexité dans le pire des cas intéressante, a été décrite. Un compromis efficace, gérant spécifiquement contraintes non-binaires et contraintes implicites, nommé sDC-2.1, a été proposé. Des expérimentations de ces algorithmes, utilisant la génération de contraintes implicites « à la volée » ont été décrites, et montré combien cette approche était prometteuse.

Les expérimentations ont également permis de mettre en évidence les faiblesses potentielles de la consistance duale, pouvant empêcher son utilisation sur des problèmes industriels. De nouvelles idées prospectives ont été proposées pour apporter des solutions à ces inconvénients.

Remerciements. Ces travaux ont été réalisés dans le cadre du projet CANAR (ANR-06-BLAN-0383-02). L'auteur tient à remercier Christian Bessière, Thierry Petit, les autres membres des projets CANAR et COCONUT, ainsi que les relecteurs anonymes pour leur aide.

Références

[1] N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20(12) :97–123, 1994.

[2] C. Bessière. *Handbook of Constraint Programming*, chapter 3 : Constraint Propagation, pages 29–84. Elsevier, 2006.

[3] C. Bessière and R. Debruyne. Theoretical analysis of singleton arc consistency and its extensions. *Artificial Intelligence*, 172(1) :29–41, 2008.

[4] C. Bessière and J.-C. Régin. Arc consistency for general constraint networks : preliminary results. In *Proceedings of IJCAI'97*, 1997.

[5] C. Bessière, J.-C. Régin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2) :165–185, 2005.

[6] F. Boussemart, F. Hemery, and C. Lecoutre. Revision ordering heuristics for the Constraint Satisfaction Problem. In *Proceedings of CPAI'04 workshop held with CP'04*, pages 29–43, 2004.

[7] F. Boussemart, F. Hemery, C. Lecoutre, and L. Saïs. Support inference for generic filtering. In *Proceedings of CP'04*, pages 721–725, 2004.

[8] R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.

[9] I.P. Gent, C. Jefferson, and I. Miguel. Watched literals for constraint propagation in Minion. In *Proceedings of CP'06*, pages 182–197, 2006.

[10] C. Lecoutre, S. Cardon, and J. Vion. Conservative Dual Consistency. In *Proceedings of AAAI'07*, pages 237–242, 2007.

[11] C. Lecoutre, S. Cardon, and J. Vion. Path Consistency by Dual Consistency. In *Proceedings of CP'2007*, 2007.

[12] C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proceedings of IJCAI'2007*, pages 125–130, 2007.

[13] C. Lecoutre and J. Vion. Enforcing Arc Consistency using Bitwise Operations. *Constraint Programming Letters*, 2 :21–35, 2008.

[14] U. Montanari. Network of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7 :95–132, 1974.

[15] C. Piette. Let the solver deal with redundancy. In *Proceedings of ICTAI'08*, volume 1, pages 67–73, 2008.

[16] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of AAAI'94*, pages 362–367, 1994.

[17] J.R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Science*, 177 :3639–3678, 2007.

[18] M. van Dongen, C. Lecoutre, and O. Roussel. Third International CSP Solvers Competition. http ://www.cril.univ-artois.fr/CPAI08, 2008.

[19] P. van Hentenryck, Y. Deville, and CM. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57 :291–321, 1992.

[20] J. Vion. Constraint Satisfaction Problem for Java. http ://cspfj.sourceforge.net/, 2006.